Optimization of “Kosaraju Algorithm” to Find the Strongly Connected Components in a Graph.

Graphs : A Representation of Data in the form of Vertices and Edges where Edges are the lines connecting Two Vertices of a Graph.

Mathematically, Graph is defined as G=(V,E), where
V=Set of Vertices {v1,v2,v3......vi}
E=Set of Edges {e1,e2,e3.....ei}.
Here, |E| can be 0(Trivial Graph) but |V| can never be 0.

Connectivity : Connectivity in an **undirected graph** means that every vertex can reach every other vertex via any path. If the graph is not connected the graph can be broken down into Connected Components.

Strong Connectivity : Strong Connectivity applies only to **directed graphs.** A directed graph is strongly connected if there is a directed path from any vertex to every other vertex.

This is same as connectivity in an undirected graph, the only difference being strong connectivity applies to directed graphs and there should be directed paths instead of just paths.Similar to connected components, a directed graph can be broken down into **Strongly Connected Components.**

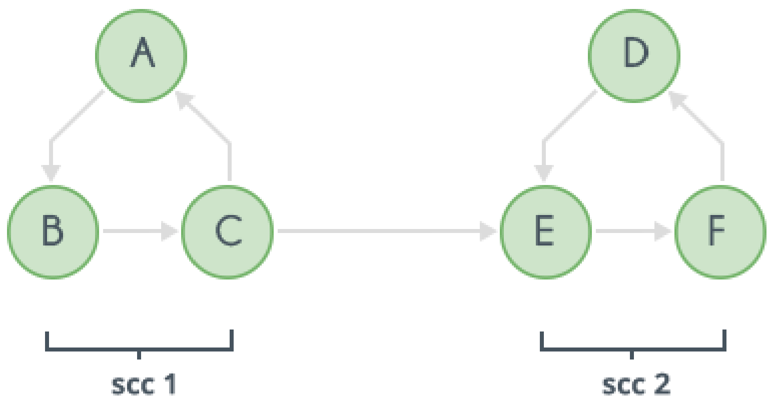

Fig 1. Original Graph

Basic/Brute Force method to find Strongly Connected Components :

Idea :

Strongly connected components(**SCC**) can be found one by one, that is first the strongly connected component including node 1 is found. Then, if node 2 is not included in the strongly connected component of node 1 , similar process which will be outlined below can be used for node 2 , else the process moves on to node 3 and so on.

So, how to find the strongly connected component which includes node 1 ?
Let there be a list which contains all nodes, these nodes will be deleted one by one once it is sure that the particular. node does not belong to the SCC of node 1 So, initially all nodes from 1 to N are in the list.Now, Let
length of list be 'L', current index be 'i', and the element at current index be 'E'.

Implementation :

1. For each of the elements 'X' at index i+1 to L, it can be checked if there is a directed path from *'X'* to *'E'* by a single **DFS O(V+E)** and if there is a directed path from *'E'* to *'X'* , again by a single **DFS O(V+E)**. If not, *'X'* can be safely deleted from the list.

2. Repeat Step 1 for next element (at next index i+1) of the list. This process needs to check whether elements at indices i+2,i+3...L have a directed path to element at index i+1. Also Check whether Reverse paths are there are not. If not, such nodes can be deleted from the list.

3. Repeat process and keep on deleting elements that must not be there in the SCC of 1.At the end List will contain the SCC's of 1.

4. To find the other Strongly Connected Components, Repeat from Step 1 on the next element(ie. 2), only if it has not already been a part of some previous Strongly Connected Component.Else process continues for next nodes.

Complexity :

The Time Complexity of the above algorithm is $O(V^3)$.

The Space Complexity of the above Algorithm is $O(V^2)$.

Kosaraju's Linear time algorithm to find Strongly Connected Components :

Condensed Component Graph : It is a graph graph with **nodes<=V and edges<=E,** in which every node is a Strongly Connected Component and there is an edge from node C to C′ if there is an edge from any node of C to any node of C′.

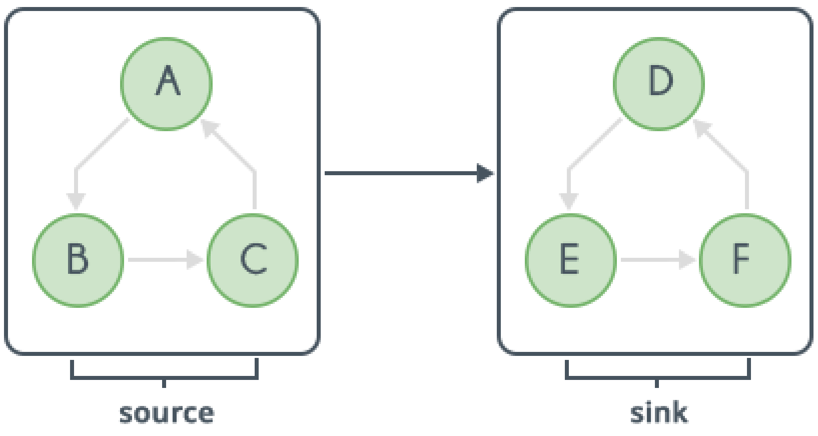

Fig 2. Condensed Component Graph

Idea :

It can be proved that Condensed Component Graph will be a **Directed Acyclic Graph(DAG).** We can prove this by the Contradiction. Suppose that there exist a cycle in Condensed Component Graph then, all the nodes in one Connected Component of the Cycle can reach to every other node of Connected Component of the Cycle, which is a contradiction.

Therefore, the Condensed Component Graph will be a **DAG.** Now, DAG a has the property that there is at least one node with no incoming edges and at least one node with no outgoing edges, namely **Source** and **Sink** respectively.It can be observed that if DFS is done from a node in Sink then only nodes in Sink SCC will get visited. On removing this Sink the graph will still remain DAG so we can repeat the same process for the new Sink node.

Property : For any two nodes **C** and **C′** of the Condensed Component Graph that share an edge, that is let (ie **C--->C′**) be an edge. The property is that the finish time of some node in C will be always higher than the finish time of all nodes of C′.

Proof: There are cases, when DFS first discovers either a node in C or a node in C′.

Case 1: When DFS first discovers a node in C :

At some time during the DFS, nodes of C′ will start getting discovered (as there is an edge from C to C′), then all nodes of C′ will be discovered (Because it is a Strongly Connected Component and will visit everything it can, before it backtracks to the node in C, from where the first visited node of was called). Therefore for this case, the finish time of some node of C will always be higher than finish time of all nodes of C′.

Case 2: When DFS first discovers a node in C' :

Now, no node of C has been discovered yet.DFS of graph will visit every node of C′ and maybe more of other Strongly Connected Component's if there is an edge from C′ to that Strongly Connected Component. Observe that now any node of C will never be discovered because there is no edge from C′ to C. Therefore, every node of C′ is already finished and any node of C has not even started yet. So clearly finish time of some node (in this case all) of C, will be higher than the finish time of all nodes of C′.

Inference :

Using this Prove we can say that Topological Sorting of condensed component graph can be done, and then some node in the leftmost Strongly Connected Component will have higher finishing time than all nodes in the Strongly Connected Component's to the right in the topological sorting.

Now the only problem left is how to find some node in the sink's Strongly Connected Component of the condensed component graph. If **The condensed component graph** is reversed, then all the **sources will become sinks** and all the **sinks will become sources.** Meanwhile the Strongly Connected Component's of the reversed graph will be same as the Strongly Connected Components of the original graph.

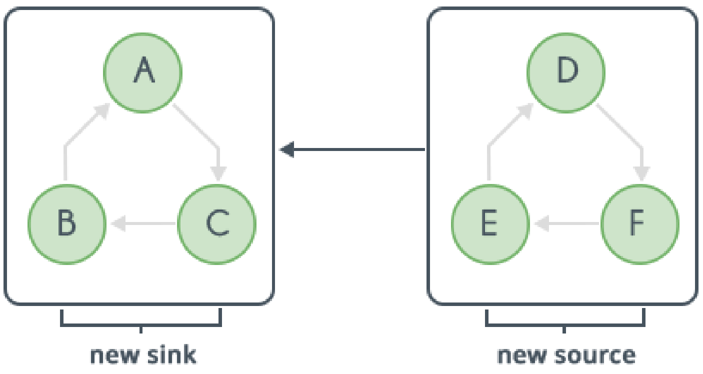

Fig 3. Reverse Condensed Component Graph

Now a DFS can be done on the new sinks, which will again lead to finding Strongly Connected Components. Now the order in which DFS on the new sinks needs to be done, is known. The order is that of **decreasing finishing times** in the of the original graph. This is because it was already proved that an edge from C to C′ in the original condensed component graph means that finish time of some node of C is always higher than finish time of all nodes of C′. So when the graph is reversed, sink will be that Strongly Connected Component in which there is a node with the highest finishing time. Since edges are reversed, a DFS from the node with highest finishing time, will visit only its own Strongly Connected Component of it. In this way all Strongly Connected Component's of Input Graph will be found.

Implementation :

1. Do a DFS on the original graph, keeping track of the finish times of each node(ie. Push Each node to a Stack when it is Finished).

2. Reverse the original graph, it can be done efficiently if data structure used to store the graph is an adjacency list.

3. Pop out a vertex from the Stack. If this vertex is a part of and any previous SCC then continue, Else do DFS on the reversed graph, with this vertex as the source vertex.

4. Repeat Step 3 until there are vertices present in the Stack.

Complexity :

The Time Complexity of the above algorithm is O(V+2E).

The Space Complexity of the above Algorithm is O(V+2E).

Optimized approach of Kosaraju's algorithm to find Strongly Connected Components :

Idea:

The idea is to Print the Strongly Connected Components always from the first discovered node of that SCC. If it is done so then, All the SCC’s to which it is acting as a Source will be visited and finally Source SCC will get printed.

Property : The first discovered node of a SCC will always be the last one to finish.

Proof: There are cases, when DFS first discovers a node which either leads to a node in C′ or a node in the same SCC or both.

Case 1: When DFS first discovers a node that leads to C’:

At some time during the DFS, nodes of C′ will start getting discovered (as there is an edge from C to C′), then all nodes of C′ will be discovered (Because it is a Strongly Connected Component and will visit everything it can, before it backtracks to the node in C, from where the first visited node of was called). So, first visited node will finish last.

Case 2: When DFS first discovers a node that leads to another node in C :

Now, no node of C has been discovered yet. DFS of graph will visit every node of C as it forms a Strongly Connected Component. Clearly first visited node will finish lastly.

Case 3: When DFS first discovers a node that leads to nodes both in C & C’:

This Case is Trivially true from the above 2 cases.

Now the only problem left is how to determine the first discovered node of a SCC? This is very simple to answer. As we know that there would be no Backedge from C′ to C (as it forms a DAG). So, all the possible Backedges will only be present in one Component. There are 2 Cases which needs to taken care of :

Case 1: When SCC consists of only 1 node:

In this case this node will not have any BackEdge and can only lead to some other SCC. So, the SCC’s associated with it will be printed first (as their first discovered node finishes) and after then this node will finish and get printed.

Case 2: When SCC consists of more than 1 node :

Since SCC consists of more than 1 node, so the first Discovered node will have some Backedges to it. So, we can determine the first discovered node with its minimum Discovery time.

Implementation :

1. Do a DFS on the original graph, keeping track of the Discovery time of each node.
2. During DFS determine the minimum Discovery time in one's SCC.
3. If a node has this minimum time then print its SCC (as this node will only be the first visited node of that SCC).

Complexity :

The Time Complexity of the above algorithm is O(V+E).

The Space Complexity of the above Algorithm is O(1).

Conclusion:

I. There is no need to keep a stack to store the finishing order of Vertices.

II. There is no need to Reverse the graph.

III. The Whole process can be done in just a single DFS scan of Graph.